Software Production Methodology Testbed Project

R. C. Tausworthe
DSN Data Systems Development Section

This article reports the history and results of a 3-1/2-year study in software development methodology. The findings of this study have become the basis for DSN software development guidelines and standard practices. The article discusses accomplishments, discoveries, problems, recommendations and future directions.

I. Introduction

Since November 1972, the DSN has been engaged in the development of a software production methodology, a set of Standard Practices which implement this methodology, and a set of languages and aids for software implementation, which together form the DSN Programming System. The software methodology development was principally an empirical bootstrapping from emerging theories in software production through a testbed implementation project into a viable modern software engineering discipline.

The testbed project was the complete redesign of an MBASIC language processor (Ref. 1) under controlled conditions which could be altered to observe effects. Design and management methods which produced favorable effects were recorded for future projects in the form of a two-volume "software methodology textbook" (Ref. 2). At this writing, both the testbed project and the textbook Volume I, exposing the methods used to effect the completion of the testbed, are complete. This article discloses the key final project guidelines, the key design concepts, the machine-independent design philosophy, the testbed methodology accomplishments and discoveries, the testbed project statistics (including team productivity), some problems not solved during this testbed activity, some recommendations for future projects, and some future needs indicated by the testbed project.

II. History

In November 1972, the Telecommunications Division and the Tracking and Data Acquisition Planning Office formed a joint team to establish a viable advanced engineering activity in development and application of a standard DSN language and the development and promulgation of engineering and technical management standards for the implementation of software for operational use in DSN facility subsystems and in DSN

technical management and administration. Later that month, this team initiated the machine-independent design of an MBASIC language processor, to be designed and documented by National Information Systems, Inc. (NIS), of Los Altos, California, to JPL standards with respect to design methodology, documentation format, level of detail, quality, and management.

A prototype version of the MBASIC processor had, at that time, just become operational (Fig. 1) on the Univac 1108 in the General-Purpose Computing Facility, having been implemented by International Computer Equipment, Inc., private consultants, and NIS. This prototype was not suitable for transfer to other host systems, was not documented to an effective level, and was probably not extendable to the intended full MBASIC language without an almost complete rewrite. Structured programming and other forms of modern software engineering, in their infancy, had not been used.

The MBASIC machine-independent design (MID) was therefore originally commissioned to fulfill the following work statement (October 1972):

(1) The design would be machine-independent down to a set of subroutines whose design would be machine-dependent.

(2) The level of design detail would extend only to that degree sufficient to permit determination of trade-offs for programming in assembly language on 16- and 32-bit hosts.

(3) Decision tables would be used to the maximum extent.

(4) Data structures and module interfaces would be specified in detail.

(5) Top-down, hierarchic, structured design principles would be applied, and documentation of the design down to five levels, plus a narrative overview, would be delivered.

Subsequent to this statement of work were several redirections of effort, the most significant of which was the transformation of the MID effort into a modern software engineering methodology testbed project, and this single redirection was the basis for all other redirections. By means of such a testbed on a software development of significant size, it was possible to formulate, implement, observe, and tune software design, documentation, coding, testing, and technical management methodologies to observe their effects in a clinical way.

The methodology instructions began in November 1972 as a set of memoranda sent to NIS; by January 1973, enough of these had been issued that, at the request of the Tracking and Data Acquisition Office, they were compiled into a "standards methodology working paper." This working paper was augmented and revised continually as the development guidelines evolved, and always kept current. It was distributed at several junctures in draft form.

In order for others to be able to use the working paper and to apply the methodologies in their own projects, it was found necessary to add tutorial material to the working paper. By June 1975, the paper had taken on the appearance of a methodology textbook, had gone through seven drafts, each including more and more tutorial detail, and had split into two volumes, methods and standards. The first volume is currently being typeset, and the second volume is drafted through most of its textual chapters and a few of its appendices.

The textbook drafts enjoyed a wide exposure to the programming community, both at JPL and outside, being used as texts for JPL seminars and classes, and for classes taught by the author and others as a graduate course to professional programmers through West Coast University. Feedback from these exposures and from members of the JPL Committee on Modern Programming (COMP), the DSN Software Management Seminar, and the DSN Programming System Steering Committee continually influenced the textbook material.

By the first part of December 1972, the first two levels of the design had been delivered and reviewed. This constituted what at the time was considered to be an "architectural overview," dealing with the technical aspects of the program procedure only. Several objections were raised with the material presented, principally with respect to the readability, detail level, and format of the design documentation.

By this time, too, a QA audit of the prototype (U1108) processor had revealed that its documentation was faulty, so the designers, over the next 3 months, were diverted from the MID until the U1108 documentation was brought up to date.

Once the U1108 documentation was up to date, there was then an intense trial and error, instruct and observe, modify and review activity over the next 6 months to develop the principal design and documentation practices which would remain in effect for the remainder of the MID project. From October 1973 through January 1974,

all flowcharts and narrative explanations of the processor down through level 3 (50 flowcharts) were prepared, reviewed, reprepared, and finally reviewed in mid-February by a formal review board.

The board concurred that design criteria (see Section III) were being fulfilled, whereupon the major design work began. Design milestones during the next 2 years are shown in Fig. 1. Figure 2 is a graphic display of the team cumulative productivity over the same period. Four of the significant milestones, to be discussed later, also are shown on the figure: a changeover of personnel, from high-level program architects to less experienced programmers, in November of 1974; the commencement of concurrent coding of the design by JPL personnel on the Caltech Decsystem-10 in March 1975; the delivery of all flowcharts and narratives in November 1975; and the formal completion of the design activity in March 1976, whereupon the MBASIC cognizant development engineer (CDE) was made responsible for the repair of minor errors detected in coding and testing the design.

III. Final Development Guidelines

The development guidelines in effect at the formal termination of the design phase were, first and foremost, that the design be hierarchically modular, limited in control connectivity to sequence, DOWHILE, WHILEDO, IFTHENELSE, and CASE (Ref. 2) structures, and readable (and developed principally) from the top down. Only one nonstructured termination form for a module was permitted, namely that for processing error messages and returns back to the MBASIC command mode.

Design documentation was in the form of flowcharts and accompanying narrative keyed to each flowchart. Additionally, tables, formats, data structures, error messages, and a glossary formed necessary reference appendices. Such documentation detail was, on a flowchart by flowchart basis, to fulfill three criteria:

(1) Adequacy for coding by remote coders without consultation, without functional ambiguity.

(2) Adequacy to assess correctness and for carrying on the design with no major redirections necessary.

(3) Adequacy of final document for use as principal sustaining document.

In order for the MBASIC CDE to review the design with respect to these three criteria, it became necessary for him to code the design on the Caltech Decsystem-10 and perform checkout measures on a flowchart by flowchart basis. These checkout measures consisted of informal top-down tests of the evolving programming dummy stubs for as yet uncoded or undesigned features, and executing the program in a special debug-monitor mode provided by the Decsystem-10.

Although such checkout tests were designed and performed informally, the guidelines for testing were explicit: submit the embryonic processor to input conditions that cause the traversal of each flowline on each flowchart newly added to the program, at least once.

All items submitted by NIS were kept under strict project control. Flowcharts and other items contained "signature control boxes," in which were placed initials of the designer, a concurring peer, and the accepting CDE; such items were given "accepted" status and so logged. Only items once accepted could be coded and tested; such modules having errors detected were changed in the log to "rework" status and returned to NIS for correction. All code was a direct translation of the flowcharts, box for box.

Items supplied by NIS on an information-only basis (lookahead design items) and items formally submitted but not yet approved by the CDE, were logged as "awaiting disposition." These modules could be coded only for use as dummy stubs, if needed. Errors detected in the use of such dummies were fed back to NIS, but no change in status resulted.

Items having been identified (as, for example, by a striped module on a flowchart) but not delivered yet in any form were marked "unseen" and logged.

Figure 2 displays the rate at which modules awaiting disposition and above were delivered (upper line), and the rate at which the modules received accepted status (lower line). In mid-1975, the status monitor of logged modules was computerized, which accounts for the appearance of increased detail during the latter part of the project.

IV. Machine-Independent Design Concepts

Because the MID was to be implementable on a range of host computers, the design had to be documented using higher-level, machine-independent descriptors of algorithms, data structures, and operating system services and interfaces. For this reason, flowcharts were chosen as the medium for expression of procedure, with narrative written to explain each flowchart, material keyed to each box on the chart. A set of precisely defined conventions for operations within flowchart boxes was adopted, and

these were inserted into the design documentation as programming standards.

These conventions were so rigorously defined that, had a compiler been available to recognize them, the translation from flowchart to code would have been automatized.

The machine-independent design extended from the top-level flowchart (called Chart 1) downwards through the program hierarchy to a set of stubs whose algorithm was felt to be potentially machine-dependent but whose interface with the remainder of the design (outside these stubs) was still machine-independent. These stubs were said to form the "environmental interface" with the host system, and were labeled "E-routines."

Figure 3 shows the separation of the MBASIC processor into the Fundamental language (that portion being the subject of Vol. I of Ref. 1) processor, herein called the MID, together with the machine-independent design of extensions to the full language (MIDX) and the machine-independent design of the compiled-code, or batch, processor (MIDB). The machine-dependent design (MDD), MDDX, and MDDB portions are machine-dependent designs needed to interface the machine-independent algorithms to the host operating system (e.g., for I/O).

In many instances, algorithms for processing MBASIC statements were deemed to be machine-dependent only because of certain constants which were likely to vary from host to host. Such algorithms were included into the MID by defining parameterized values for the constants, giving these a special notation (mnemonic name prefixed by "%") so as to be discernible to implementors.

To implement MBASIC on a host, the flowcharts comprising the MID can be coded immediately in host assembly language, manually, or perhaps using a set of portable macros (Ref. 3). The remainder of the job is to perform the machine-dependent design of the E-routines, and then to code and test these. Machine-dependent variation in performance of each implementation is to be tolerated only as permitted within the program specification (Ref. 1).

V. Testbed Methodology Accomplishments

Among the accomplishments of this testbed activity may be listed the firm establishment of procedures forming the central core of current DSN standard practices, which advocate the use of top-down structured design methods and sound engineering practices. The project demonstrated that the design documentation was, in fact, adequate for coding by remote coders; that the top-down approach can accommodate personnel changeovers with a minimum project impact; and that, in fact, junior design-level personnel can replace senior architect-level personnel without adverse effect once the major program structure and higher-level design decisions have been worked out.

As may be noted in Fig. 2, the pure-design rate (before rework starts coming back from the CDE) is much higher than a design-plus-correctness-assessment rate is apt to be. This is evidenced by the "sprint" of modules generated at the beginning and after new personnel were brought on board. This demonstrates that design to establish a program architecture is feasible without the need for coding when detail correctness is not an issue.

However, as demonstrated by the increased slope of the acceptance line, concurrent coding is an aid to correctness assessment early in a project and a necessity (the jagged falloff) in its latter stages.

The curves in Fig. 2 demonstrate that meaningful quantitative monitors of programming progress exist, based on public programming practices. The public programming practices here consisted of regular submittals of flowcharts, narratives, tables, etc. (completed and lookahead), and the logging of such items by completion category. The missing item needed to make early predictions of costs and schedules was only the top asymptote, not known in the present case until almost mid-project (July 1975).

Recommendations to improve progress monitors appear in Section IX.

VI. Testbed Project Statistics

The MID specification consists of about 950 flowcharts (plus narrative) and E-routine interface descriptions, plus tables, the glossary, etc. By considering all documentation items uniformly distributed over the 950 module deliverables, one may compute that, over approximately 1250 man-days, approximately 3/4 of a module was designed and documented per man-day from the go-ahead review in February 1974 until project completion in March 1976. The low productivity rate is principally a reflection of the difficulty of doing an extremely intricate processing task in a machine-independent way, as opposed to doing the same task in a machine-dependent way.

In corroboration of this, the Decsystem-10 figures are 1.7 modules per man-day for a 525 man-day total,

including design, coding, debugging, and testing. The MID coding took only 100 man-days to code but another 175 man-days coding and testing to get the MID error-free. Of significant note is that only 18% of the time was spent in coding and debugging, and 82% in design.

The total expenditure for the development and first implementation (MID + MDD) was about 2050 man-days, or about 8.2 man-years. This represents about 13 lines of code per man-day.

However, the total next-time effort may be expected to be only 625 man-days, or about 2.5 man-years (525 + 100 man-days). The implementation on the next host is therefore only about 30% of the initial development effort, or about 45 lines of code per man-day.

VII. Methodology Discoveries

At the beginning of the project, a strict top-down discipline was a stated requirement, imposed on the design team to verify claims in the literature (Ref. 4) that strict top-down methods were profitable. It was soon discovered that, while the submission of formal design items, documentation, and delivery of codable flowcharts from the top-down was of great merit, strict adherence to the top-down development discipline was having an adverse effect both on achieving a good design and on securing a high initial correctness.

On modifying the development guidelines, it was learned that a lookahead design effort to supplement the top-down method and to establish the program architecture was of great benefit as a precursor to the formal top-down detailed design, documentation, and subsequent coding. Such lookahead, it was found, did not need to be absolutely correct in detail, so long as the designers' intentions were recorded, work tasks identified, and the overall program size estimated. Coding to assess correctness by the designers was *not* needed, and, in fact, would probably have been a hindrance if done. Some coding might have been useful to make certain performance tradeoffs, however.

Once the formal top-down development began, however, coding was found to be needed, not only to prove program correctness but also to reinforce the designers psychologically.

Coding and checking design items soon after submission and project acceptance was found to avert the "correctness paranoia" syndrome which tends to set in when the design gets too large for mental retention of intricate details. Rather, when designers *know* what they have produced thus far is correct, they stop worrying about it and go on to more productive things.

The design effort got off to a slow slow start, it seemed, for several reasons, principal of which was a general lack of familiarity with the top-down, modular, hierarchic, structured programming methodology. Indeed, at that time, much of the methodology had not been developed yet, and was in the process of being developed as a result of this design work. However, it was found, as new personnel were brought in, that they, too, required a period of training in the methodology before they could do an effective job.

Another aspect of the design task, which not only decreased initial productivity, but which was generally believed to be unachievable to begin with, was the requirement for a machine-independent design. Structuring a program for machine-independence in all but a set of environmental stubs was clearly not a production task, but one which required applied research. The philosophies and methods developed in this area will be the subject of a future article.

However, in retrospect, it now appears that at least half of the algorithms in an entire implementation can fall into this machine-independent design category. Moreover, these algorithms form the entire upper-level structure of the MBASIC processor. The machine-dependent items are relegated to environmental interfacing stubs whose implementations do not require personnel with language-processor-design skills.

Because the architectural overview and overall processing philosophy of the program were, unfortunately, not made an early part of the formal design supplied to the Decsystem-10 and MODCOMP-II implementation teams (JPL and contractor personnel), we found that there was a great need for this type of information early in the coding and testing phases.

The first correctness test procedures were very formal and very detailed. It was soon discovered that such procedures are difficult to write, lengthy, and require great expenditures of time, and at no discernible increase in level of correctness over less formal procedures, given standard guidelines. The standard guideline was to execute the program modules under test using the entire program (or major segment) as a driver, with dummy stubs for modules not yet coded and with path monitors adequate to trace the control flow; to select input data to cause the program to traverse each flowline in each module under test at least once, plus extreme-value data, out-of-bounds

data, and randomly chosen valid data; to run the program with its stubs using the selected input; and to exercise human judgment whether or not the trace information indicated that the intended functions were being accomplished.

Selection of detailed procedures and test data, as well as the criteria for judgment of correctness, were left to the discretion of the individual implementors. Testing using the standard guideline proved to be very efficient, quick (with respect to more formal methods), and thorough.

The method used to log flowcharts, described earlier, proved very useful in gauging the current status of delivered items. Because an entire lookahead was not done prior to the commencement of the formal detailed design, however, the ultimate number of modules was unknown until quite late. Thus, incremental progress could be seen on a regular basis, but estimated percentage of completion was unavailable. Had a complete architectural (lookahead) design been done and from this, a work breakdown structure generated, we can now see in retrospect that it would have been possible to monitor percentage completion figures right from the beginning of the formal phase, and to report progress as milestones defined in the work breakdown structure.

VIII. Documentation Discoveries

The principal documentation of the MID processor was in the form of flowcharts and narratives. Flowcharts were hand-drawn using templates with ANSII-standard symbols, labeled by typewriter. All flowcharts conformed to rigid structured programming and detail requirements. There was a separate page or two of narrative for each chart, supplied to explain each step in the charted procedures, its significance, and its rationale. Such documentation was initiated in rough draft form by the NIS designers, then typed and drafted by clerical personnel before submission to JPL.

It was found that this documentation format was excellent with respect to communications between designers and coders. We had achieved the goal that flowcharts were codable by remote personnel without consultation. The quality of the documentation was generally excellent, although there was a relatively high volume of paper for the amount of information it contained.

However, the documentation medium proved to be cumbersome in the sense that the design process tended to get in series with the limited clerical personnel available and assigned to the task. For this reason, turn-around time for effecting changes was slow. In the end, we wound up working much of the time from red-lined charts returned to NIS in "rework" status, having discovered errors by coding and testing and repaired these ourselves in red-line form.

The level of documentation, with respect to detail, seemed to follow the inequality

DESIGNER < SUSTAINING < CO-LOCATED CODER < REMOTE CODER

The level required [so-called Class A (Ref. 4), adequate for remote coders] was therefore deemed adequate for later sustaining of the design. However, the differential between the level of detail required by the designer and that needed by others was very great. Designers could get by very well with little detail, and getting them to supply this detail, at times, was somewhat of a problem. Writing down all the details was not a creative job but something, we learned, that could be provided by relatively junior design-level personnel.

Generally, then, the higher-level architects developed the major algorithms in flowcharted sketches, which the more junior designers then supplied with details and narratives. In so doing, the junior-level designers learned enough of the design and methodology that, midway in the project, they were able to replace their more expensive colleagues altogether. The senior people were free for more creative and less drudging tasks.

The fact that there is a rather routine, drudging part of the documentation task, which tends to avert creative people from doing it, is an indication that much can be done to improve the design medium to bring it into closer alignment with what is needed by coders.

For one thing, it was found that once an overview of the processing, data structuring, and architecture of the program was gained by coders, they tended not to need the detailed narratives supplied. Rather, they coded directly from the charts, which were explicit enough for that task without narrative recourse. Hence, probably a lot of work went into creation of narratives which could have been avoided had a suitable overview been insisted upon and provided in the beginning. The overview, incidentally, was the last-produced piece of documentation.

The overview was requested (but not insisted upon) early but was felt at the time not to be a high-priority item. Developing it, too, must have been viewed a drudging job by the architects, who then would have had to write it themselves if it were to be among the first-

produced deliverables. Some recommendations for future projects are contained in the next section.

The final documentation discovery discussed here is felt to be of major significance: comment-free code. Alarming as it sounds, coding the MID (and MDD as well) had, by project standards, almost no narration in the assembly code listings. The comment fields of the assembly listing were limited to cross-reference annotations of the flowchart number, title, and box number. This cross-referencing made it very easy to correspond lines of code with actions in the design for coding, debugging, and later QA audits. Only when special coding was used, or when it was otherwise not clear how a certain function on a flowchart was achieved by the code, were narrative comments permitted. Since the level of detail required produced explicit coding implications, virtually no such narratives were needed.

The practice permitted fast, routine coding; coders did not have to make up comments, except in rare instances, and made no design decisions (in coding the MID at least). But more importantly, it virtually forced the *design* medium (flowcharts + narrative) to also become the *debugging* medium. Any corrections to be made were first identified, then entered on the flowchart (red-lined), and then inserted into the code, rather than vice-versa. Because of this, the documentation was always kept up to date as a natural consequence of the program development, not as an after-the-fact, error-prone process. A great deal of expense had gone into producing quality documentation, and comment-free code proved a way of not subverting that quality by creating a processor maintainable at the code level alone.

IX. Recommendations

In future testbed activities (specifically, the MBASIC extension to full language capability), we intend to, and recommend that other projects also, perform an entire lookahead design (no coding and detailed correctness not at issue) down to a sufficient level to establish the architecture and the number of modules and schedule to an accuracy goal of 10%. The architecture documentation will consist predominantly of hand-drawn flowcharts and other graphics, including sketches of major data structures (narratives also permitted but not required). The idea is to permit the architect to carry the design through all the way to that point needed to size the job, to gauge the number of flowcharts in the final design to 10%. We currently estimate that about 20-25% of the total development time will be spent in this activity.

Additionally, the designer will then be asked to produce a work breakdown structure defining interim milestones, tasks, personnel assignments, and determination of critical-path items.

During this architectural phase, the evolving flowchart module tree will be recorded, as well as estimated completion percentages of other architectural tasks, at regular (biweekly) intervals.

Once the architecture is complete, and a Software Definition Document written and reviewed per DSN Standard Practice, the formal development of detailed, high-initial-correctness flowcharts, narratives, and other items will begin.

The concurrent coding effort will begin early in the post-architecture phase so as to reduce design-error risk, but not so early as to constrain the design prerogatives.

Comment fields in the code listings will continue to contain documentation references only, except in special circumstances. Coding will continue to be a direct translation of the flowcharts, in 1 to 1 correspondence, box for box.

Testing for correctness (as opposed to acceptance testing) will continue to be performed on an informal basis but in conformance with the formal guideline given earlier.

Because the MIDX design is to extend the MID, the same documentation guidelines will be in effect (for uniformity). However, overview material will be insisted upon early.

X. Problems not Solved by the Testbed Methodology

Perhaps the most noticeable failure of the testbed was in the area of minimizing problems due to non-colocation between design and coding teams, principal of which was the inability to maintain the desired visibility into the partial-progress status of the design team. Regular progress reporting was unenforceable; submitted reports, when they came, generally provided no quantifiable status information. Because status monitors were faulty, we were unable to predict schedules. It is hoped that better status monitors will be forthcoming in the MIDX follow-on because of the explicit architectural phase and work breakdown structures to be generated.

As mentioned earlier, the documentation medium (more than the documentation format) proved inflexible and expensive, and required a comparatively long turn-around time to correct even minor bugs, cosmetics, or oversights.

The clerical burden of supplying drudging narrative detail to support flowcharts tended to "burn out" design team members, especially those with the more senior capability. Because of the fear of having to renarrate (added drudgery) items found by later coding to be in error, and by emphasis placed on original correctness in the design, there seemed to be a correctness paranoia. This led to hiding of design items, or reluctance to deliver flowcharts and narratives on a piecemeal basis. This countered the "public programming" goal and decreased visibility into how much of the design we actually had at any particular point in time. We hope that the separate architectural phase (during which correctness in minor details is not at issue), the use of concurrent coding to check design details early, and the improved status monitors will avert future difficulties of this type.

XI. Future Needs Indicated by Testbed Project

Probably the single thing most needed to raise productivity at this point is a more flexible, easier-to-maintain, less expensive but equally descriptive and graphic form of program documentation. The documentation produced by designers needs to be sufficient for remote coders as a natural outcome of the documentation method; that is, the documentation method needs to narrow the gap between what the designer wants to produce and what the remote coder actually needs. (We refuse to accept designers doing the coding themselves as a method of achieving this, as it tends to produce software not maintainable by individuals other than the originators.)

The second major need is that for better project visibility producing methods during the design and implementation. Suitable methods for cost and schedule prediction will probably result when this need is fulfilled.

Perhaps the best way to increase the level of public programming (and thereby, project visibility) is to merge the design and documentation media via automation. If design media are computer-based, processing a design into readable and needed documentation can potentially be done automatically—fulfilling the first need above—and at the same time, can provide a viable status probing capability—fulfilling the second need also.

To avert human fallibility in the informal correctness testing and to speed up the path-identification and corresponding input data generation, there is a need for computer aid. The generation of tests and test data conforming to the earlier guideline is almost a clerical task in itself, tantamount to tabulation of various paths in the design and determination of data (and stub designs) to drive the coded program through those paths. Although human judgment may be needed at points in this procedure, certainly the computer is capable of scanning the design, analyzing and tabulating paths and conditions to be met for the human then to consider.

Other aids can readily be wished for. The creation of such development supporting software must always be guided by expectation or demonstration of feasibility, cost, and potential gain.

XII. Summary

The methodology developed as a result of this testbed has become the foundation of the current DSN software development guidelines and standard practices. The quality of the testbed design and of the documentation produced by it is unmistakable. There is room for yet more improvement, however, chiefly in the areas of increased productivity and project manageability. Both of these enhancements can be achieved by putting the computer to work to solve the very problems people have in creating programs for the computer.

At the beginning of this project, the development of quality software was truly an art masterable by only a few of its practitioners. The application of the methodology used and derived by this testbed demonstrates that production programming may be a passing art form. Such a passage is not lamentable, however, for it will be replaced by an effective engineering discipline. Whereas artforms are generally mastered by a few but appreciated by many, the engineering discipline will be both practiced and appreciated by an entire community of adherents.

References

1. *MBASIC*, Vol. I—Fundamentals, Vol. II—Appendices, Jet Propulsion Laboratory, Pasadena, California, 1973 (JPL internal document).
2. Tausworthe, Robert C., *Standardized Development of Computer Software*, Vol. I—Methods, Vol. II—Standards, Jet Propulsion Laboratory, June 1976 (JPL internal document).
3. Riggins, M. C., "Portability of the MBASIC Machine Independent Design," *DSN Prog. Report 42-24*, Jet Propulsion Laboratory, Dec. 15, 1974, pp. 100-106.
4. Tausworthe, Robert C., *Standard Classifications of Software Documentation*, Technical Memorandum 33-756, Jet Propulsion Laboratory, Pasadena, California, Jan. 15, 1976.

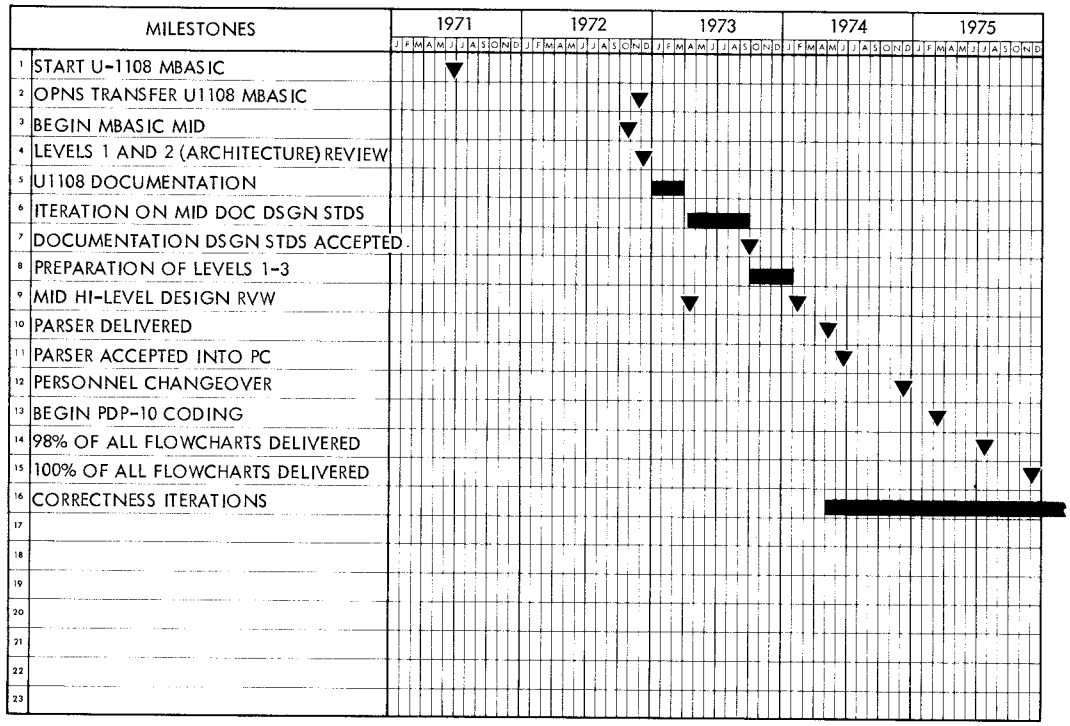

Fig. 1. MBASIC development history

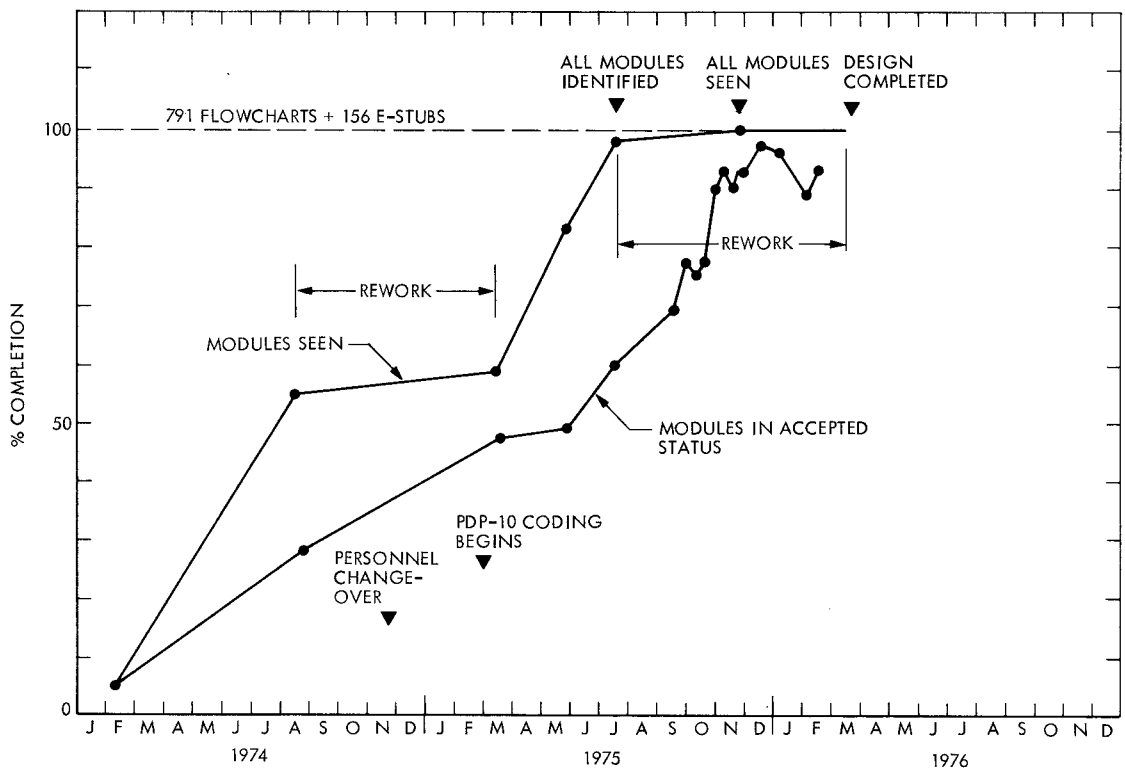

Fig. 2. MBASIC-MID module development history

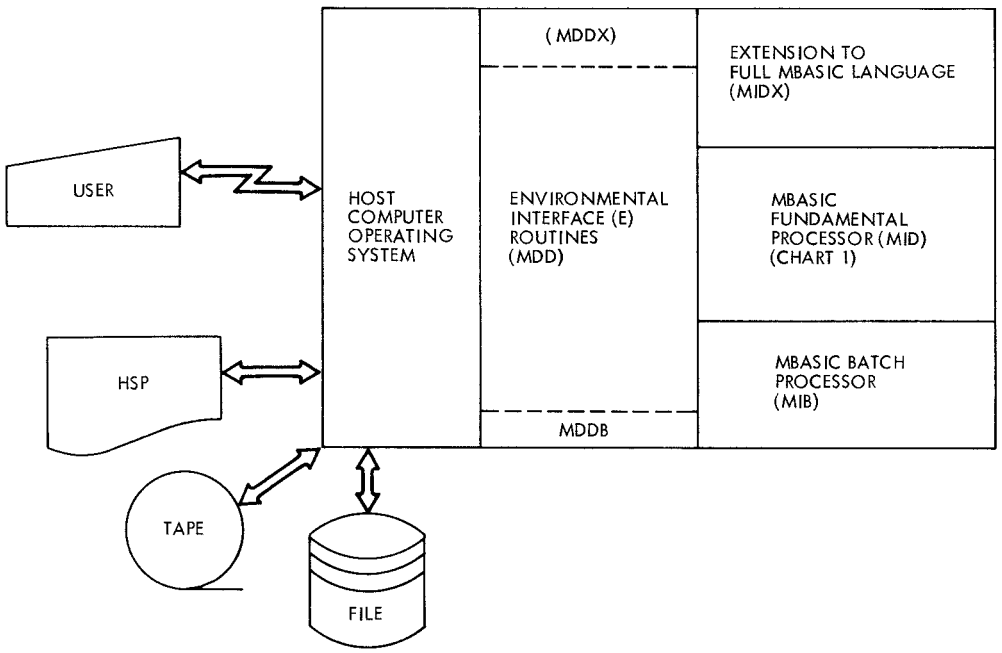

Fig. 3. MBASIC machine-independent design configuration